SPRAL_SSIDS
Sparse Symmetric Indefinite Direct Solver
Fortran User Guide

This package solves one or more sets of $n \times n$ sparse **symmetric** equations $AX = B$ using a multifrontal method on an **NVIDIA GPU**. The following cases are covered:

1. A is **indefinite**. `SSIDS` computes the sparse factorization

$$A = PLD(PL)^T$$

where P is a permutation matrix, L is unit lower triangular, and D is block diagonal with blocks of size 1×1 and 2×2.

2. A is **positive definite**. `SSIDS` computes the **sparse Cholesky factorization**

$$A = PL(PL)^T$$

where P is a permutation matrix and L is lower triangular. *However, as `SSIDS` is designed primarily for indefinite systems, this may be slower than a dedicated Cholesky solver.*

`SSIDS` returns bit-compatible results.

An option exists to scale the matrix. In this case, the factorization of the scaled matrix $\overline{A} = SAS$ is computed, where S is a diagonal scaling matrix.

Jonathan Hogg (STFC Rutherford Appleton Laboratory)
Evgueni Ovtchinnikov (STFC Rutherford Appleton Laboratory)
Jennifer Scott (STFC Rutherford Appleton Laboratory)

Major version history

2014-03-17 Version 1.0.0 Initial release

1 Installation

Please see the SPRAL install documentation. In particular note that:

- A CUDA compiler (`nvcc`) is required.
- A METIS library is required.
- A BLAS library is required (in addition to CUBLAS).

2 Usage overview

2.1 Calling sequences

Access to the package requires a `USE` statement

```
use spral_ssids
```

The following procedures are available to the user:

- `ssids_analyse()` accepts the matrix data in compressed sparse column format and optionally checks it for duplicates and out-of-range entries. The user may supply an elimination order; otherwise one is generated. Using this elimination order, `ssids_analyse()` analyses the sparsity pattern of the matrix and prepares the data structures for the factorization.
- `ssids_analyse_coord()` is an alternative to `ssids_analyse()` that may be used if the user has the matrix data in coordinate format. Again, the user may supply an elimination order; otherwise one is generated. `ssids_analyse_coord()` checks the matrix data for duplicates and out-of-range entries, stores it in compressed sparse column format and then proceeds in the same way as `ssids_analyse()`.
- `ssids_factor()` uses the data structures set up by `ssids_analyse()` to compute a sparse factorization. More than one call to `ssids_factor()` may follow a call to `ssids_analyse()` (allowing more than one matrix with the same sparsity pattern but different numerical values to be factorized without multiple calls to `ssids_analyse()`). An option exists to scale the matrix.
- `ssids_solve()` uses the computed factors generated by `ssids_factor()` to solve systems $AX = B$ for one or more right-hand sides B. Multiple calls to `ssids_solve()` may follow a call to `ssids_factor()`. An option is available to perform a partial solution.
- `ssids_free()` should be called after all other calls are complete for a problem (including after an error return that does not allow the computation to continue). It frees memory referenced by components of the derived types. This routine also allows created by either `ssids_analyse() or ssids_factorize()` to be freed separately.

In addition, the following routines may be called:

- `ssids_enquire_posdef()` may be called in the positive-definite case to obtain the pivots used.
- `ssids_enquire_indef()` may be called in the indefinite case to obtain the pivot sequence used by the factorization and the entries of D^{-1}.
- `ssids_alter()` may be called in the indefinite case to alter the entries of D^{-1}. Note that this means that $PLD(PL)^T$ is no longer a factorization of A.

2.2 Derived types

For each problem, the user must employ the derived types defined by the package to declare scalars of the types `ssids_options`, `ssids_inform`, `ssids_akeep`, and `ssids_fkeep`. The following pseudo-code illustrates this.

```
use spral_ssids
...
type (ssids_options) :: options
type (ssids_inform) :: inform
type (ssids_akeep) :: akeep
type (ssids_fkeep) :: fkeep
...
```

The components of `ssids_options` and `ssids_inform` are explained in Sections 5.1 and 5.2. The components of `ssids_akeep` and `ssids_fkeep` are used to pass data between the subroutines of the package and must not be altered by the user.

Figure 1: Data format example matrix

$$\begin{pmatrix} 1.1 & 2.2 & & 3.3 & \\ 2.2 & & 4.4 & & \\ & 4.4 & 5.5 & & 6.6 \\ 3.3 & & & 7.7 & 8.8 \\ & & 6.6 & 8.8 & 9.9 \end{pmatrix}$$

2.3 Achieving bit-compatibility

Care has been taken to ensure bit-compatibility is achieved using this solver. That is, consecutive runs with the same data on the same machine produces exactly the same solution.

2.4 Optional arguments

We use square brackets `[ ]` to indicate *optional* arguments. In each call, optional arguments follow the argument `inform`. Since we reserve the right to add additional optional arguments in future releases of the code, **we strongly recommend that all optional arguments be called by keyword, not by position**.

2.5 Integer, real and package types

`INTEGER` denotes default `INTEGER` and `INTEGER(long)` denotes `INTEGER(kind=selected_int_kind(18))`. `REAL` denotes double precision real. We also use the term **package type** to mean the same.

2.6 Data formats

2.6.1 Compressed Sparse Column (CSC) Format

This standard data format consists of the following data:

```
integer                   :: n      ! size of matrix
integer, size(n+1)        :: ptr    ! column pointers
integer, size(ptr(n+1)-1) :: row    ! row indices
real,    size(ptr(n+1)-1) :: val    ! numerical values
```

Non-zero matrix entries are ordered by increasing column index and stored in the arrays `row(:)` and `val(:)` such that `row(k)` holds the row number and `val(k)` holds the value of the `k`-th entry. The `ptr(:)` array stores column pointers such that `ptr(i)` is the position in `row(:)` and `val(:)` of the first entry in the `i`-th column, and `ptr(n+1)` is one more than the total number of entries. Entries that are zero, including those on the diagonal, need not be specified.

If this format is used, `SSIDS` requires only the lower triangular entries of A, and there should be no duplicate entries. If the `check` argument to `ssids_analyse()` is `.true.`, out-of-range entries (including those in the upper triangle) will be discarded and any duplicates will be summed.

To illustrate the CSC format, the following arrays describe the matrix shown in Figure 1.

```
n = 5
ptr(1:6) = (/ 1,             4,   5,        7,        9,    10 /)
row(1:9) = (/ 1,   2,   4,   3,   3,   5,   4,   5,   5 /)
val(1:9) = (/ 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9 /)
```

2.6.2 Coordinate Format

This standard data format consists of the following data:

```
integer           :: n     ! size of matrix
integer           :: ne    ! number of non-zero entries
integer, size(ne) :: row   ! row indices
integer, size(ne) :: col   ! column indices
real,    size(ne) :: val   ! numerical values
```

The arrays should be set such that the k-th entry is in row row(k) and column col(k) with value val(k). Entries that are zero, including those on the diagonal, need not be specified.

If this format is used, SSIDS requires that each entry of A should be present **only** in the lower *or* upper triangular part. Entries present in both will be summed, as will any duplicate entries. Out-of-range entries are ignored.

To illustrate the coordinate format, the following arrays describe the matrix shown in Figure 1.

```
n = 5
ne = 9
row(1:9) = (/ 1,   2,   3,   4,   3,   5,   4,   5,   5 /)
col(1:9) = (/ 1,   1,   2,   1,   3,   3,   4,   4,   5 /)
val(1:9) = (/ 1.1, 2.2, 4.4, 3.3, 5.5, 6.6, 7.7, 8.8, 9.9 /)
```

3 Basic Subroutines

3.1 ssids_analyse() and ssids_analyse_coord()

To analyse the sparsity pattern and prepare for the factorization,

for Compressed Sparse Column (CSC) format:

```
call ssids_analyse(check,n,ptr,row,akeep,options,inform[,order,val])
```

for Coordinate format:

```
call ssids_analyse_coord(n,ne,row,col,akeep,options,inform[,order,val])
```

Matrix data should be supplied as described in Section 2.6. As the package uses CSC format internally, ssids_analyse() with checking disabled provides the most efficient interface.

check is an INTENT(IN) scalar of type LOGICAL. If set to .true. the matrix data is checked for errors and the cleaned matrix (duplicates are summed and out-of-range entries discarded) is stored in akeep. Otherwise, for data in CSC format, no checking of the matrix data is carried out and ptr(:) and row(:) must be passed unchanged to the factorization routines. Checking is always performed when the coordinate format is used.

n, ptr(:), row(:) are INTENT(IN) variables of type INTEGER specifying the lower triangular part of A in CSC format (see Section 2.6.1).

n, ne, row(:), col(:) are INTENT(IN) variables of type INTEGER specifying the lower (or upper) triangular part of A in coordinate format (see Section 2.6.2).

akeep is an INTENT(OUT) scalar of type ssids_akeep. It is used to hold data about the problem being solved and must be passed unchanged to the other subroutines.

options is an INTENT(IN) scalar of type ssids_options. Its components specify the algorithmic options used by the subroutine, as explained in Section 5.1.

inform is an INTENT(OUT) scalar of type ssids_inform. Its components provide information about the execution of the subroutine, as explained in Section 5.2.

order(:) is an optional INTENT(INOUT) array of type INTEGER and size n. If options%ordering=0, order(:) must be present and order(i) must hold the position of variable i in the elimination order. On exit, order(:) contains the elimination order that ssids_factor() will be given (it is passed to these routines as part of akeep); this order may give slightly more fill-in than the user-supplied order and, in the indefinite case, may be modified by ssids_factor() to maintain numerical stability.

val(:) is an optional INTENT(IN) array of package type that must hold the numerical values of the entries of the matrix, as described in Section 2.6. val(:) must be present if a matching-based elimination ordering is required (options%ordering=2), and is otherwise ignored.

3.2 ssids_factor()

To factorize the matrix,

```
call ssids_factor(posdef,val,akeep,fkeep,options,inform[,scale,ptr,row])
```

`posdef` is an `INTENT(IN)` scalar of type `LOGICAL` that must be set to `.true.` if the matrix is positive-definite, and `.false.` if it is indefinite.

`val(:)` is an `INTENT(IN)` array of package type that must hold the numerical values of the entries of the matrix, as described in Section 2.6.

`akeep` is an `INTENT(IN)` scalar of type `ssids_akeep` that must be unchanged since the call to `ssids_analyse()` or `ssids_analyse_coord()`.

`fkeep` is an `INTENT(INOUT)` scalar of type `ssids_fkeep`. It is used to hold data about the problem being solved and must be passed unchanged to the other subroutines.

`options` is an `INTENT(IN)` scalar of type `ssids_options`. Its components specify the algorithmic options used by the subroutine, as explained in Section 5.1.

`inform` is an `INTENT(OUT)` scalar of type `ssids_inform`. Its components provide information about the execution of the subroutine, as explained in Section 5.2.

`scale(:)` is an optional `INTENT(INOUT)` array of type `REAL` and size `n`. If present and `options%scaling=0`, it must contain the diagonal entries of the scaling matrix S. On exit, `scale(i)` will contain the `i`-th diagonal entry of the scaling matrix S.

`ptr(:)` and `row(:)` are optional `INTENT(IN)` arrays of type `INTEGER`. They are only accessed if `ssids_analyse()` was called with `check` set to `.false.`. In this case, they must both be present and must be unchanged since that call.

3.3 ssids_solve()

To solve the linear system $AX = B$, after a call to `ssids_factor()`,

for a single right-hand side:

```
call ssids_solve(x1,akeep,fkeep,options,inform[,job])
```

for one or more right-hand sides:

```
call ssids_solve(nrhs,x,ldx,akeep,fkeep,options,inform[,job])
```

Partial solutions may be performed by appropriately setting the optional parameter `job`.

`x1(:)` is an `INTENT(INOUT)` array of package type and size `n`. It must be set such that `x1(i)` holds the component of the right-hand side for variables i. On exit, `x1(i)` holds the solution for variable i.

`nrhs` is an `INTENT(IN)` scalar of type `INTEGER` that holds the number of right-hand sides.

`x(:,:)` is an `INTENT(INOUT)` array of package type with extents `ldx` and `nrhs`. It must be set so that `x(i,j)` holds the component of the right-hand side for variable i to the j-th system. On exit, `x(i,j)` holds the solution for variable i to the j-th system.

`ldx` is an `INTENT(IN)` scalar of type `INTEGER` that must be set to the first extent of array `x(:)`.

`akeep` is an `INTENT(IN)` scalar of type `ssids_akeep` that must be unchanged since the last call to `ssids_factor()`.

`fkeep` is an `INTENT(IN)` scalar of type `ssids_fkeep` that must be unchanged since the last call to `ssids_factor()`.

`options` is an `INTENT(IN)` scalar of type `ssids_options`. Its components specify the algorithmic options used by the subroutine, as explained in Section 5.1.

`inform` is an `INTENT(OUT)` scalar of type `ssids_inform`. Its components provide information about the execution of the subroutine, as explained in Section 5.2.

`job` is an optional `INTENT(IN)` scalar of type `INTEGER`. If absent, $AX = B$ is solved. In the positive-definite case, the Cholesky factorization that has been computed may be expressed in the form

$$SAS = (PL)(PL)^T$$

where P is a permutation matrix and L is lower triangular. In the indefinite case, the factorization that has been computed may be expressed in the form

$$SAS = (PL)D(PL)^T$$

where P is a permutation matrix, L is unit lower triangular, and D is block diagonal with blocks of order 1 and 2. S is a diagonal scaling matrix (S is equal to the identity, if `options%scaling=0` and `scale` is not present on the last call to `ssids_factor()`). A partial solution may be computed by setting `job` to have one of the following values:

`1` for solving $PLX = SB$

`2` for solving $DX = B$ (indefinite case only)

`3` for solving $(PL)^T S^{-1} X = B$

`4` for solving $D(PL)^T S^{-1} X = B$ (indefinite case only)

3.4 ssids_free()

To free memory and resources,

allocated by `ssids_analyse()` or `ssids_analyse_coord()`:

```
call ssids_free(akeep,cuda_error)
```

allocated by `ssids_factor()`:

```
call ssids_free(fkeep,cuda_error)
```

both at once:

```
call ssids_free(akeep,fkeep,cuda_error)
```

Once all other calls are complete for a problem or after an error return that does not allow the computation to continue, a call should be made to free memory and CUDA resources allocated by `SSIDS` and associated with the derived data types `akeep` and/or `fkeep` using calls to `ssids_free()`.

`akeep` is an `INTENT(INOUT)` scalar of type `ssids_akeep` that must be passed unchanged. On exit, allocatable components will have been deallocated, and CUDA resources released.

`fkeep` is an `INTENT(INOUT)` scalar of type `ssids_fkeep` that must be passed unchanged. On exit, allocatable components will have been deallocated, and CUDA resources released.

`cuda_error` is an `INTENT(OUT)` scalar of type default integer. On exit, a non-zero value gives a CUDA error code. This may indicate either a failure to deallocate GPU memory, or a pre-existing CUDA error condition. Note that due to the asynchronous nature of GPU execution, the reported error may have a cause errors external to `SSIDS`.

4 Advanced subroutines

4.1 ssids_enquire_posdef()

To obtain the matrix D following a positive-definite factorization,

```
call ssids_enquire_posdef(akeep,fkeep,options,inform,d)
```

`akeep` is an `INTENT(IN)` scalar of type `ssids_akeep` that must be unchanged since the call to `ssids_factor()`.

`fkeep` is an `INTENT(IN)` scalar of type `ssids_fkeep` that must be unchanged since the call to `ssids_factor()`.

`options` is an `INTENT(IN)` scalar of type `ssids_options`. Its components specify the algorithmic options used by the subroutine, as explained in Section 5.1.

`inform` is an `INTENT(OUT)` scalar of type `ssids_inform`. Its components provide information about the execution of the subroutine, as explained in Section 5.2.

`d(:)` is an `INTENT(OUT)` array of type `REAL` and size `n`. The i-th diagonal entry of D will be placed in `d(i)`.

4.2 ssids_enquire_indef()

To obtain the matrix D^{-1} and/or the pivot order following an indefinite factorization,

```
call ssids_enquire_indef(akeep,fkeep,options,inform[,piv_order,d])
```

`akeep` is an `INTENT(IN)` scalar of type `ssids_akeep` that must be unchanged since the call to `ssids_factor()`.

`fkeep` is an `INTENT(IN)` scalar of type `ssids_fkeep` that must be unchanged since the call to `ssids_factor()`.

`options` is an `INTENT(IN)` scalar of type `ssids_options`. Its components specify the algorithmic options used by the subroutine, as explained in Section 5.1.

`inform` is an `INTENT(OUT)` scalar of type `ssids_inform`. Its components provide information about the execution of the subroutine, as explained in Section 5.2.

`piv_order(:)` is an optional `INTENT(OUT)` array of type `INTEGER` and size `n`. If present, on exit `|piv_order(i)|` gives the position of variable i in the pivot order. The sign will be positive if i is a 1×1 pivot, and negative if i is part of a 2×2 pivot.

`d(:,:)` is an optional `INTENT(OUT)` array of package type with extents `2` and `n`. If present, on exit diagonal entries of D^{-1} will be placed in `d(1,i)`, $i = 1, 2, \ldots, n$, the off-diagonal entries of D^{-1} will be placed in `d(2,i)`, $i = 1, 2, \ldots, n-1$, and `d(2,n)` will be set to zero.

4.3 ssids_alter()

To alter $\mathbf{D}^{-1}$ following an indefinite factorization,

```
call ssids_alter(d,akeep,fkeep,options,inform)
```

Note that this routine is not compatabile with the option `options.presolve=1`.

`d(:,:)` is an `INTENT(IN)` array of package type with extents `2` and `n`. The diagonal entries of D^{-1} will be altered to `d(1,i)`, $i = 1, 2, \ldots, n$, and the off-diagonal entries will be altered to `d(2,i)`, $i = 1, 2, \ldots, n-1$ (and $PLD(PL)^T$ will no longer be a factorization of A).

`akeep` is an `INTENT(IN)` scalar of type `ssids_akeep` that must be unchanged since the call to `ssids_factor()`.

`fkeep` is an `INTENT(INOUT)` scalar of type `ssids_fkeep` that must be unchanged since the call to `ssids_factor()`.

5 Derived types

5.1 ssids_options

The derived data type `ssids_options` is used to specify the options used within `SSIDS`. The components, that are automatically given default values in the definition of the type, are:

Printing options

`print_level` is a scalar of type `INTEGER` that is used to control the level of printing. The different levels are:

- < 0 No printing.
- `= 0` Error and warning messages only.
- `= 1` As 0, plus basic diagnostic printing.
- > 1 As 1, plus some additional diagnostic printing.

The default is `print_level`=0.

`unit_diagnostics` is a scalar of type `INTEGER` that holds the unit number for diagnostic printing. Printing is suppressed if `unit_diagnostics`< 0. The default is `unit_diagnostics`=6.

`unit_error` is a scalar of type `INTEGER` that holds the unit number for error messages. Printing of error messages is suppressed if `unit_error`<0. The default is `unit_error`=6.

`unit_warning` is a scalar of type `INTEGER` that holds the unit number for warning messages. Printing of warning messages is suppressed if `unit_warning`<0. The default is `unit_warning`=6.

Options used by `ssids_analyse()` and `ssids_analyse_coord()`

`ordering` is a scalar of type `INTEGER`. If set to `0`, the user must supply an elimination order in `order(:)`; otherwise an elimination order will be computed by `ssids_analyse()` or `ssids_analyse_coord()`. The options are:

- `0` User-supplied ordering is used.
- `1` METIS ordering with default settings is used.
- `2` A matching-based elimination ordering is computed (the Hungarian algorithm is used to identify large off-diagonal entries. A restricted METIS ordering is then used that forces these on to the subdiagonal). This option should only be chosen for indefinite systems. A scaling is also computed that may be used in `ssids_factor()` (see `options%scaling` below).

The default is `ordering`=1. **Restriction:** `ordering`=0, 1, 2.

`nemin` is a scalar of type `INTEGER` that controls node amalgamation. Two neighbours in the elimination tree are merged if they both involve fewer than `nemin` eliminations. The default is `nemin`=8. The default is used if `nemin`<1.

Options used by `ssids_factor()`

`scaling` is a scalar of type default `INTEGER` that controls the use of scaling. The available options are:

- $\leq$ `0` No scaling (if `scale(:)` is not present), or user-supplied scaling (if `scale(:)` is present).
- = `1` Compute a scaling using a weighted bipartite matching via the Hungarian Algorithm (`MC64` algorithm).
- = `2` Compute a scaling using a weighted bipartite matching via the Auction Algorithm (may be lower quality than that computed using the Hungarian Algorithm, but can be considerably faster).
- = `3` A matching-based ordering has been generated during the analyse phase using `options%ordering` = 2. Use the scaling generated as a side-effect of this process. The scaling will be the same as that generated with `options%scaling` = 1 if the matrix values have not changed. This option will generate an error if a matching-based ordering was not used.
- $\geq$ `4` Compute a scaling using the norm-equilibration algorithm of Ruiz.

The default is `scaling`=0.

Options used by `ssids_factor()` with `posdef=.false.` (A indefinite)

`action` is a scalar of type default `LOGICAL`. If the matrix is found to be singular (has rank less than the number of non-empty rows), the computation continues after issuing a warning if `action` has the value `.true.` or terminates with an error if it has the value `.false.`. The default is `action=.true.`.

`u` is a scalar of type `REAL` that holds the relative pivot tolerance u. The default is `u=0.01`. Values outside the range $[0, 0.5]$ are treated as the closest value in that range.

Options used by `ssids_factor()` and `ssids_solve()`

`use_gpu_solve` is a scalar of type `LOGICAL` that controls whether to use the CPU or GPU for the `ssids_solve()`. If `.true.`, the GPU is used, but some more advanced features are not available (see the description of the `job` parameter in Section 3.3). Setting `use_gpu_solve=.false.` is only compatible with `options%presolve=0`. The default value is `use_gpu_solve=.true.`.

`presolve` is a scalar of type `INTEGER` that controls the amount of extra work performed during factorization to accelerate the solve. It can take the following values:

- `0` Minimal work is performed during `ssids_factor()` to prepare for the solve.
- `1` The explicit inverse of the `nelim`×`nelim` block in each supernode is precalculated during `ssids_factor()` (where `nelim` is the number of variables eliminated at that supernode). As the matrix L is overwritten, the routine `ssids_alter()` cannot be used. This option is not compatible with `options%use_gpu_solve=.false.`.

The default option is `presolve=0`.

5.2 ssids_inform

The derived data type `ssids_inform` is used to hold parameters that give information about the progress and needs of the algorithm. The components of `ssids_inform` (in alphabetical order) are:

`flag` is a scalar of type `INTEGER` that gives the exit status of the algorithm (details in Section 6).

`matrix_dup` is a scalar of type `INTEGER`. On exit from `ssids_analyse()` with `check` set to `.true.` or from `ssids_analyse_coord()`, it holds the number of duplicate entries that were found and summed.

`matrix_missing_diag` is a scalar of type `INTEGER`. On exit from `ssids_analyse()` with `check` set to `.true.`, or from `ssids_analyse_coord()`, it holds the number of diagonal entries without an explicitly provided value.

`matrix_outrange` is a scalar of type `INTEGER`. On exit from `ssids_analyse()` with `check` set to `.true.` or from `ssids_analyse_coord()`, it holds the number of out-of-range entries that were found and discarded.

`matrix_rank` is scalar of type `INTEGER`. On exit from `ssids_analyse()` and `ssids_analyse_coord()`, it holds the structural rank of A, if available (otherwise, it is set to `n`). On exit from `ssids_factor()`, it holds the computed rank of the factorized matrix.

`maxdepth` is a scalar of type `INTEGER`. On exit from `ssids_analyse()` or `ssids_analyse_coord()`, it holds the maximum depth of the assembly tree.

`maxfront` is a scalar of type `INTEGER`. On exit from `ssids_analyse()` or `ssids_analyse_coord()`, it holds the maximum front size in the positive-definite case (or in the indefinite case with the same pivot sequence). On exit from `ssids_factor()`, it holds the maximum front size.

`num_delay` is scalar of type `INTEGER`. On exit from `ssids_factor()`, it holds the number of eliminations that were delayed, that is, the total number of fully-summed variables that were passed to the father node because of stability considerations. If a variable is passed further up the tree, it will be counted again.

num_factor is scalar of type INTEGER(long). On exit from ssids_analyse() or ssids_analyse_coord(), it holds the number of entries that will be in the factor L in the positive-definite case (or in the indefinite case with the same pivot sequence). On exit from ssids_factor(), it holds the actual number of entries in the factor L. In the indefinite case, 2n entries of D^{-1} are also held.

num_flops is scalar of type INTEGER(long). On exit from ssids_analyse() or ssids_analyse_coord(), it holds the number of floating-point operations that will be needed to perform the factorization in the positive-definite case (or in the indefinite case with the same pivot sequence). On exit from ssids_factor(), it holds the number of floating-point operations performed.

num_neg is a scalar of type INTEGER. On exit from ssids_factor(), it holds the number of negative eigenvalues of the matrix D.

num_sup is a scalar of type INTEGER. On exit from ssids_analyse() or ssids_analyse_coord(), it holds the number of supernodes in the problem.

num_two is scalar of type INTEGER. On exit from ssids_factor(), it holds the number of 2×2 pivots used by the factorization, that is, the number of 2×2 blocks in D.

stat is a scalar of type INTEGER. In the event of an allocation or deallocation error, it holds the Fortran stat parameter if it is available (and is set to 0 otherwise).

cublas_error is a scalar of type INTEGER. In the event of an error return from the CUBLAS library, it holds the error code returned (and is 0 otherwise).

cuda_error is a scalar of type INTEGER. In the event of a CUDA error, it holds the error code returned (and is 0 otherwise). Note that due to the asynchronous nature of GPU execution, the reported error may have a cause errors external to SSIDS.

6 Return codes

A successful return from a subroutine in the package is indicated by inform%flag having the value zero. A negative value is associated with an error message that by default will be output on unit options%unit_error.

Possible negative values are:

−1 An error has been made in the sequence of calls (this includes calling a subroutine after an error that cannot be recovered from).

−2 Returned by ssids_analyse() and ssids_analyse_coord() if n<0. Also returned by ssids_analyse_coord() if ne<1.

−3 Returned by ssids_analyse() if there is an error in ptr(:).

−4 Returned by ssids_analyse() if all the variable indices in one or more columns are out-of-range. Also returned by ssids_analyse_coord() if all entries are out-of-range.

−5 Returned by ssids_factor() if posdef=.false. and options%action = .false. when the matrix is found to be singular. The user may reset the matrix values in val(:) and recall ssids_factor().

−6 Returned by ssids_factor() if posdef=.true. and the matrix is found to be not positive definite. This may be because the Hungarian scaling determined that the matrix was structurally singular. The user may reset the matrix values in val(:) and recall ssids_factor().

−7 Returned by ssids_factor() if ssids_analyse() was called with check set to .false. but ptr(:) and/or row(:) is not present.

−8 Returned by ssids_analyse() and ssids_analyse_coord() if options%ordering is out-of-range, or options%ordering=0 and the user has either failed to provide an elimination order or an error has been found in the user-supplied elimination order (as supplied in order(:)).

−9 Returned by ssids_analyse() and ssids_analyse_coord() if options%ordering=2, but val(:) was not supplied.

−10 Returned by `ssids_solve()` if there is an error in the size of array `x(:,:)` (that is, `ldx`<`n` or `nrhs`<1). The user may reset `ldx` and/or `nrhs` and recall `ssids_solve()`.

−11 Returned by `ssids_solve()` if `job` is out-of-range. The user may reset `job` and recall `ssids_solve()`.

−12 Returned by `spral_ssids_solve()` and `spral_ssids_alter()` if the selected combination of `options%use_gpu_solve` and `options%presolve` are not compatible with the requested operation.

−13 Returned by `ssids_enquire_posdef()` if `posdef`=`.false.` on the last call to `ssids_factor()`.

−14 Returned by `ssids_enquire_indef()` if `posdef`=`.true.` on the last call to `ssids_factor()`.

−15 Returned by `ssids_factor()` if `options%scaling`=3 but a matching based ordering was not used during the call to `ssids_analyse()` or `ssids_analyse_coord()` (i.e. was called with `options%ordering`≠2).

−50 Allocation error. If available, the `stat` parameter is returned in `inform%stat`.

−51 CUDA error. The CUDA error return value is returned in `inform%cuda_error`.

−52 CUBLAS error. The CUBLAS error return value is returned in `inform%cublas_error`.

A positive value of `inform%flag` is used to warn the user that the input matrix data may be faulty or that the subroutine cannot guarantee the solution obtained. Possible values are:

+1 Returned by `ssids_analyse()` and `ssids_analyse_coord()` if out-of-range variable indices found. Any such entries are ignored and the computation continues. `inform%matrix_outrange` is set to the number of such entries.

+2 Returned by `ssids_analyse()` and `ssids_analyse_coord()` if duplicated indices found. Duplicates are recorded and the corresponding entries are summed. `inform%matrix_dup` is set to the number of such entries.

+3 Returned by `ssids_analyse()` and `ssids_analyse_coord()` if both out-of-range and duplicated variable indices found.

+4 Returned by `ssids_analyse()` and `ssids_analyse_coord()` if one and more diagonal entries of A is missing.

+5 Returned by `ssids_analyse()` and `ssids_analyse_coord()` if one and more diagonal entries of A is missing and out-of-range and/or duplicated variable indices have been found.

+6 Returned by `ssids_analyse()` and `ssids_analyse_coord()` if A is found be (structurally) singular. This will overwrite any of the above warnings.

+7 Returned by `ssids_factor()` if `options%action` is set to `.true.` and the matrix is found to be (structurally or numerically) singular.

+8 Returned by `ssids_factor()` if `options%ordering`=2 (i.e. a matching-based ordering was used) but the associated scaling was not (i.e. `options%scaling`≠ 3).

7 Method

`ssids_analyse()` and `ssids_analyse_coord()`

If `check` is set to `.true.` on the call to `ssids_analyse()` or if `ssids_analyse_coord()` is called, the user-supplied matrix data is checked for errors. The cleaned integer matrix data (duplicates are summed and out-of-range indices discarded) is stored in `akeep`. The use of checking is optional on a call to `ssids_analyse()` as it incurs both time and memory overheads. However, it is recommended since the behaviour of the other routines in the package is unpredictable if duplicates and/or out-of-range variable indices are entered.

If the user has supplied an elimination order it is checked for errors. Otherwise, an elimination order is generated by the package. The elimination order is used to construct an assembly tree. On exit from `ssids_analyse()` (and `ssids_analyse_coord()`), `order(:)` is set so that `order(i)` holds the position of variable i in the elimination order. If an ordering was supplied by the user, this order may differ, but will be equivalent in terms of fill-in.

ssids_factor()

ssids_factor() optionally computes a scaling and then performs the numerical factorization. The user must specify whether or not the matrix is positive definite. If posdef is set to .true., no pivoting is performed and the computation will terminate with an error if a non-positive pivot is encountered.

The factorization uses the assembly tree that was set up by the analyse phase. At each node, entries from A and, if it is not a leaf node, the generated elements and any delayed pivots from its child nodes must be assembled. Separate kernels handle each of these.

The kernel that performs the assembly from the child nodes considers one parent-child assembly at a time. Each generated element from a child is divided into a number of tiles, and a thread block launched to assemble each tile into a dense submatrix using a simple mapping array to determine the destination row and column of each entry. Bit-compatibility is achieved by ensuring the child entries are always assembled in the same order.

A dense partial factorization of the fully summed columns is then performed. The fully summed columns are split into a number of tiles that are each handled by an associated block. Factorization proceeds one column of tiles at a time. The pivoting condition is chosen to ensure that all entries of L have absolute value less than $\mathtt{u}^{-1}$. This limits the growth of the entries of the D factor and ensures that any solves will be backwards stable. The details are described in [1].

If a pivot candidate does not pass the pivot tests at a node, it is delayed to its parent node, where further elimination operations may make it acceptable. Delaying pivots leads to additional fill-in and floating-point operations beyond that predicted by ssids_analyse() (and ssids_analyse_coord()), and may result in additional memory allocations being required. The number of delayed pivots can often be reduced by using appropriate scaling.

At each non-root node, the majority of the floating-point operations involve the formation of the generated element. This is handled by a single dedicated kernel; again, see [1] for details.

At the end of the factorization, data structures for use in future calls to ssids_solve() are prepared. If options%presolve=1, the block of L corresponding to the eliminated variables is explicitly inverted to accelerate future calls to ssids_solve() at the cost of making ssids_factor() slower.

ssids_solve()

If options%use_gpu_solve=.false., data is moved to the CPU if required and the BLAS calls are used to perform a solve using the assembly tree and factors generated on previous calls.

Otherwise, the solve is conducted on the GPU in a similar fashion. If options%presolve=0, custom GPU implementations of _trsv() and _gemv() are used to handle multiple independent operations. If multiple right-hand sides are to be solved for, the single right-hand side solve is looped over. If options%presolve=1, _trsv() can be replaced by the much more parallel (and hence faster) _gemv(). In this case multiple right-hand sides are handled at the same time.

References

[1] J.D. Hogg, E. Ovtchinnikov and J.A. Scott. (2014). A sparse symmetric indefinite direct solver for GPU architectures. RAL Technical Report. RAL-P-2014-0xx, to appear.

8 Example

Suppose we wish to factorize the matrix

$$A = \begin{pmatrix} 2. & 1. & & & \\ 1. & 4. & 1. & & 1. \\ & 1. & 3. & 2. & \\ & & 2. & 0. & \\ & 1. & & & 2. \end{pmatrix}$$

and then solve for the right-hand side

$$B = \begin{pmatrix} 4. \\ 17. \\ 19. \\ 6. \\ 12. \end{pmatrix}.$$

The following code may be used.

```
! examples/Fortran/ssids.f90 - Example code for SPRAL_SSIDS package
program ssids_example
   use spral_ssids
   implicit none

   ! Derived types
   type (ssids_akeep)   :: akeep
   type (ssids_fkeep)   :: fkeep
   type (ssids_options) :: options
   type (ssids_inform)  :: inform

   ! Parameters
   integer, parameter :: wp = kind(0.0d0)

   ! Matrix data
   logical :: posdef
   integer :: n, ptr(6), row(8)
   real(wp) :: val(8)

   ! Other variables
   integer :: piv_order(5), cuda_error
   real(wp) :: x(5)
   logical :: check

   ! Data for matrix:
   ! ( 2  1         )
   ! ( 1  4  1    1 )
   ! (    1  3  2   )
   ! (       2      )
   ! (    1       2 )
   posdef = .false.
   n = 5
   ptr(1:n+1)        = (/ 1,        3,             6,      8,8,   9 /)
   row(1:ptr(n+1)-1) = (/ 1,   2,   2,   3,   5,   3,   4,   5   /)
   val(1:ptr(n+1)-1) = (/ 2.0, 1.0, 4.0, 1.0, 1.0, 3.0, 2.0, 2.0 /)

   ! The right-hand side with solution (1.0, 2.0, 3.0, 4.0, 5.0)
   x(1:n) = (/ 4.0, 17.0, 19.0, 6.0, 12.0 /)

   ! Perform analyse and factorise with data checking
   check = .true.
   call ssids_analyse(check, n, ptr, row, akeep, options, inform)
   if(inform%flag<0) go to 100
   call ssids_factor(posdef, val, akeep, fkeep, options, inform)
   if(inform%flag<0) go to 100

   ! Solve
   call ssids_solve(x,akeep,fkeep,options,inform)
   if(inform%flag<0) go to 100
```

```
   write(*,'(/a,/,(3es18.10))') ' The computed solution is:', x(1:n)

   ! Determine and print the pivot order
   call ssids_enquire_indef(akeep, fkeep, options, inform, piv_order=piv_order)
   write(*,"(a,10i5)") ' Pivot order:', piv_order(1:n)

   100 continue
   call ssids_free(akeep, fkeep, cuda_error)

end program ssids_example
```

This produces the following output:

```
 Warning from ssids_analyse. Warning flag =   4
 one or more diagonal entries is missing

 The computed solution is:
  1.0000000000E+00  2.0000000000E+00  3.0000000000E+00
  4.0000000000E+00  5.0000000000E+00
 Pivot order:   4    5   -2   -1    3
```

Funded by EPSRC grant EP/J010553/1